Rust Book - 6.1: Enums and Pattern Matching

#rust Enums and Pattern Matching

Enums allow you to define a type by enumerating its possible values.

Rust's enums are most similar to*algebraic data types*in functional languages, such as F#, OCaml, and Haskell.

E.g. IP address… we can **enumerate** all possible values, which is where enumeration gets its name.
These are known as the **variants** of the enum:

```
enum IpAddrKind { V4, V6, }
struct IpAddr {
    kind: IpAddrKind,
    address: String,
}
```

```
enum IpAddr { V4(String), V6(String),}
let home = IpAddr::V4(String::from("127.0.0.1"));
let loopback = IpAddr::V6(String::from("::1"));
```

If we wanted to store `V4` addresses as four `u8` values but still express `V6` addresses as one `String` value, we wouldn't be able to with a struct. Enums handle this case with ease:

```
enum IpAddr {
    V4(u8, u8, u8, u8),
    V6(String),
}
let home = IpAddr::V4(127, 0, 0, 1);
let loopback = IpAddr::V6(String::from("::1"));
```

TheOptionEnum and Its Advantages Over Null Values

The `Option` type is used in many places because it encodes the very common scenario in which a value could be something or it could be nothing. Expressing this concept in terms of the type system means the compiler can check whether you've handled all the cases you should be handling; this functionality can prevent bugs that are extremely common in other programming languages.

Rust doesn't have the null feature that many other languages have.*Null*is a value that means there is no value there. In languages with null, variables can always be in one of two states: null or not-null.

the concept that null is trying to express is still a useful one: a null is a value that is currently invalid or absent for some reason.

```
enum Option<T> {
    Some(T),
    None,
}
```

```
let some_number = Some(5);
let some_string = Some("a string");
let absent_number: Option<i32> = None;
```

When we have a `Some` value, we know that a value is present and the value is held within the `Some`. When we have a `None` value, in some sense, it means the same thing as null: we don't have a valid value.

because `Option<T>` and `T` (where `T` can be any type) are different types, the compiler

won't let us use an `Option<T>` value as if it were definitely a valid value. For example, this code won't compile because it's trying to add an `i8` to an `Option<i8>`:

you have to convert an `Option<T>` to aTbefore you can perform `T` operations with it. Generally, this helps catch one of the most common issues with null: assuming that something isn't null when it actually is.

Not having to worry about incorrectly assuming a not-null value helps you to be more confident in your code. In order to have a value that can possibly be null, you must explicitly opt in by making the type of that value `Option<T>`. Then, when you use that value, you are required to explicitly handle the case when the value is null. Everywhere that a value has a type that isn't an `Option<T>`, you*can*safely assume that the value isn't null. This was a deliberate design decision for Rust to limit null's pervasiveness and increase the safety of Rust code.

in order to use anOption<T>value, you want to have code that will handle each variant. You want some code that will run only when you have aSome(T)value, and this code is allowed to use the innerT. You want some other code to run if you have aNonevalue, and that code doesn't have aTvalue available.

The `match` Control Flow Operator

`match` allows you to compare a value against a series of patterns and then execute code based on which pattern matches. Patterns can be made up of literal values, variable names, wildcards, and many other things

```
enum Coin {
   Penny, Nickel, Dime, Quarter,
}

fn value_in_cents(coin: Coin) -> u32 {
    match coin {
        Coin::Penny => {
            println!("Lucky penny!");
            1
```

```
        },
        Coin::Nickel => 5,
        Coin::Dime => 10,
        Coin::Quarter => 25,
    }
}
```

Patterns that Bind to Values

```
enum UsState {
    Alabama,
    Alaska,
    // --snip--
}

enum Coin {
    Penny,
    Nickel,
    Dime,
    Quarter(UsState),
}

fn value_in_cents(coin: Coin) -> u32 {
    match coin {
        Coin::Penny => 1,
        Coin::Nickel => 5,
        Coin::Dime => 10,
        Coin::Quarter(state) => {
            println!("State quarter from {:?}!", state);
            25
        },
    }
}
```

Matching with `Option<T>`

…a function that takes an `Option<i32>` and, if there's a value inside, adds 1 to that value. If there isn't a value inside, the function should return the `None` value and not attempt to perform any operations.

```
fn plus_one(x: Option<i32>) -> Option<i32> {
    match x {
        None => None,
        Some(i) => Some(i + 1),
    }
}
let five = Some(5);
let six = plus_one(five);
let none = plus_one(None);
```

You'll see this pattern a lot in Rust code: `match` against an enum, bind a variable to the data inside, and then execute code based on it.

Matches in Rust are *exhaustive* : we must exhaust every last possibility in order for the code to be valid.

Matches in Rust are*exhaustive*: we must exhaust every last possibility in order for the code to be valid.

The_Placeholder

The_Placeholder

Rust also has a pattern we can use when we don't want to list all possible values.

```
let some_u8_value = 0u8;
match some_u8_value {
    1 => println!("one"),
```

```
    3 => println!("three"),
    5 => println!("five"),
    7 => println!("seven"),
    _ => (),
}
```

Concise Control Flow withif let

Concise Control Flow withif let

Theif letsyntax lets you combineifandletinto a less verbose way to handle values that match one pattern while ignoring the rest.

```
let some_u8_value = Some(0u8);
match some_u8_value {
    Some(3) => println!("three"),
    _ => (),
}
```

```
```

```
```